H

hw1-doc

0 0 https://gitlab02.cs.stonybrook.edu/cse320/hw1-d

Correct typos "most-significant" to "least-significant".
Eugene Stark authored 4 days ago 24c80fe4

Name	Last commit	Last update
README.md	Correct typos "most-significant" to "least-significant".	4 days ago

README.md

Homework 1 - CSE 320 - Spring 2019

Professor Eugene Stark

Due Date: Friday 09/20/2019 @ 11:59pm

Read the entire doc before you start

Introduction

In this assignment, you will write a command line utility to serialize a tree of files and directories to a sequence of bytes and to deserialize such a sequence of bytes to re-create the original tree. Such a utility could be used, for example, to "transplant" a tree of files and directories from one place to another. The Unix `cpio` utility is an example of a full-featured utility of this type. The goal of this homework is to familiarize yourself with C programming, with a focus on input/output, strings in C, bitwise manipulations, and the use of pointers.

For all assignments in this course, you **MUST NOT** put any of the functions that you write into the `main.c` file. The file `main.c` **MUST ONLY** contain `#include` s, local `#define` s and the `main` function. The reason for this restriction has to do with our use of the Criterion library to test your code. Beyond this, you may have as many or as few additional `.c` files in the `src` directory as you wish. Also, you may declare as many or as few headers as you wish. Note, however, that header and `.c` files distributed with the assignment base code often contain a comment at the beginning which states that they are not to be modified. **PLEASE** take note of these comments and do not modify any such files, as they will be replaced by the original versions during grading. For this assignment, we use `template.c` as the example file containing our helper functions. This file may be modified as you wish.

> 😱 Array indexing ('**A[]**') is not allowed in this assignment. You **MUST USE** pointer arithmetic instead. All necessary arrays are declared in the `const.h` header file. You **MUST USE** these arrays. **DO NOT** create your own arrays. We **WILL** check for this.

Getting Started

Fetch base code for `hw1` as described in `hw0` . You can find it at this link: https://gitlab02.cs.stonybrook.edu/cse320/hw1.

Both repos will probably have a file named `.gitlab-ci.yml` with different contents. Simply merging these files will cause a merge conflict. To avoid this, we will merge the repos using a flag so that the `.gitlab-ci.yml` found in the `hw1` repo will replace the `hw0` version. To merge, use this command:

```
git merge -m "Merging HW1_CODE" HW1_CODE/master --strategy-option=theirs
```

Here is the structure of the base code:

```
.
├── .gitlab-ci.yml
└── hw1
    ├── include
    │   ├── const.h
    │   ├── debug.h
    │   └── transplant.h
    ├── Makefile
    ├── rsrc
    │   ├── testdir
    │   │   ├── dir
    │   │   │   ├── goodbye
    │   │   │   └── hello1
    │   │   └── hello
    │   └── test_in.bin
    ├── src
    │   ├── main.c
    │   └── transplant.c
    └── tests
        └── hw1_tests.c
```

 Reference for pointers: https://beej.us/guide/bgc/html/multi/pointers.html.

 Reference for command line arguments: https://beej.us/guide/bgc/html/multi/morestuff.html#clargs.

Note: All commands from here on are assumed to be run from the `hw1` directory.

A Note about Program Output

What a program does and does not print is VERY important. In the UNIX world stringing together programs with piping and scripting is commonplace. Although combining programs in this way is extremely powerful, it means that each program must not print extraneous output. For example, you would expect `ls` to output a list of files in a directory and nothing else. Similarly, your program must follow the specifications for normal operation. One part of our grading of this assignment will be to check whether your program produces EXACTLY the specified output. If your program produces output that deviates from the specifications, even in a minor way, or if it produces extraneous output that was not part of the specifications, it will adversely impact your grade in a significant way, so pay close attention.

Use the debug macro `debug` (described in the 320 reference document in the Piazza resources section) for any other program output or messages you many need while coding (e.g. debugging output).

Part 1: Program Operation and Argument Validation

In this part, you will write a function to validate the arguments passed to your program via the command line. Your program will treat arguments as follows:

- If no flags are provided, you will display the usage and return with an `EXIT_FAILURE` return code
- If the `-h` flag is provided, you will display the usage for the program and exit with an `EXIT_SUCCESS` return code.
- If the `-s` flag is provided, you will perform serialization: traversing a tree of files and directories and emitting them in serialized form to `stdout`, exiting with `EXIT_SUCCESS` on success and `EXIT_FAILURE` on any error.
- If the `-d` flag is provided, you will perform deserialization: reading serialized data from `stdin` and reconstructing the tree of files and directories it describes, exiting with `EXIT_SUCCESS` on success and `EXIT_FAILURE` on any error.

 `EXIT_SUCCESS` and `EXIT_FAILURE` are macros defined in `<stdlib.h>` which represent success and failure return codes respectively.

 `stdin`, `stdout`, and `stderr` are special I/O "streams" that are automatically opened at the start of execution for all programs and do not need to be reopened.

Some of these operations will also need other command line arguments which are described in each part of the assignment. The two usage

scenarios for this program are:

```
Usage: bin/transplant -h [any other number or type of arguments]
    -h       Help: displays this help menu
Usage: bin/transplant -s|-d [-c] [-p DIR]
```

The `-s|-d` means that one or the other of `-s` or `-d` may be specified. The `[-c]` means that `-c` is an optional parameter. Likewise `[-p DIR]` means that `-p` may optionally be specified, in which case it is immediately followed by a parameter `DIR`.

Should this parameter not be present, then the current working directory is used as the source for serialization or target for deserialization.

A valid invocation of the program implies that the following hold about the command-line arguments:

- All positional arguments (`-h|-s|-d`) come before the any optional arguments (`-c` or `-p`). The optional arguments may come in any order after the positional ones.
- If the `-h` flag is provided, it is the first positional argument after the program name and any other arguments that follow are ignored.

  > If the `-h` flag is *not* specified, then exactly one of `-s`, or `-d` must be specified.

- If an option requires a parameter, the corresponding parameter must be provided (e.g. `-p` must always be followed by a pathname DIR).

For example, the following are a subset of the possible valid argument combinations:

- `$ bin/transplant -h ...`
- `$ bin/transplant -s`
- `$ bin/huff -d -p /tmp -c`

> 😱 The `...` means that all arguments, if any, are to be ignored; e.g. the usage `bin/transplant -h -x -y BLAHBLAHBLAH -z` is equivalent to `bin/transplant -h`.

Some examples of invalid combinations would be:

- `$ bin/transplant -s -c`
- `$ bin/transplant -d -p`
- `$ bin/transplant -d -p -c`

> 😱 You may use only "raw" `argc` and `argv` for argument parsing and validation. Using any libraries that parse command line arguments (e.g. `getopt`) is prohibited.
>
> 😱 Any libraries that help you parse strings are prohibited as well (`string.h`, `ctype.h`, etc). *This is intentional and will help you practice parsing strings and manipulating pointers.*
>
> 😱 You **MAY NOT** use dynamic memory allocation in this assignment (i.e. `malloc`, `realloc`, `calloc`, `mmap`, etc.).

NOTE: The `make` command compiles the `transplant` executable into the `bin` folder. Assume all commands in this doc are run from from the `hw1` directory of your repo.

Required Validate Arguments Function

In `const.h`, you will find the following function prototype (function declaration) already declared for you. You **MUST** implement this function as part of the assignment.

```
/**
 * @brief Validates command line arguments passed to the program.
 * @details This function will validate all the arguments passed to the
 * program, returning 0 if validation succeeds and -1 if validation fails.
 * Upon successful return, the selected program options will be set in the
 * global variable "global_options", where they will be accessible
 * elsewhere in the program.
 *
 * @param argc The number of arguments passed to the program from the CLI.
 * @param argv The argument strings passed to the program from the CLI.
 * @return 0 if validation succeeds and -1 if validation fails.
```

```
 * Refer to the homework document for the effects of this function on
 * global variables.
 * @modifies global variable "global_options" to contain a bitmap representing
 * the selected options.
 */
int validargs(int argc, char **argv);
```

 This function must be implemented as specified as it will be tested and graded independently. **It should always return -- the USAGE macro should never be called from validargs.**

The `validargs` function should return -1 if there is any form of failure. This includes, but is not limited to:

- Invalid number of arguments (too few or too many)
- Invalid ordering of arguments
- A missing parameter to an option that requires one (e.g. `-p` with no `DIR` specification).

The `global_options` variable of type `int` is used to record the mode of operation (i.e. help/serialize/deserialize) of the program, as well as whether the "clobber" parameter `-c` has been specified. This is done as follows:

- If the `-h` flag is specified, the least significant bit (bit 0) is 1.
- The second-least-significant bit (bit 1) is 1 if `-s` is passed (i.e. the user wants serialization mode).
- The third-least-significant bit (bit 2) is 1 if `-d` is passed (i.e. the user wants deserialization mode).
- The fourth-least-significant bit (bit 3) is 1 if `-c` is passed (i.e. the user wants to "clobber" existing files).

If `validargs` returns -1 indicating failure, your program must call `USAGE(program_name, return_code)` and return `EXIT_FAILURE`. **Once again, `validargs` must always return, and therefore it must not call the `USAGE(program_name, return_code)` macro itself. That should be done in `main`.**

If `validargs` sets the least-significant bit of `global_options` to 1 (i.e. the `-h` flag was passed), your program must call `USAGE(program_name, return_code)` and return `EXIT_SUCCESS`.

 The `USAGE(program_name, return_code)` macro is already defined for you in `const.h`.

If validargs returns 0, then your program must perform either serialization or deserialization, depending on whether `-s` or `-d` was specified. For serialization, your program is to traverse the tree of files and directories to be serialized (more on that later) and the serialized data is to be written as a sequence of bytes to `stdout`. For deserialization, the data to be deserialized is to be read as a sequence of bytes from `stdin` and your program is to recreate the tree of files and directories (more on that later) specified by this sequence of bytes.

Upon successful completion, your program should exit with exit status `EXIT_SUCCESS`; otherwise, in case of an error it should exit with exit status `EXIT_FAILURE`.

 Remember `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in `<stdlib.h>`. Also note, `EXIT_SUCCESS` is 0 and `EXIT_FAILURE` is 1.

 We suggest that you create functions for each of the operations defined in this document. Writing modular code will help you isolate and fix problems.

Sample validargs Execution

The following are examples of `global_options` settings for given inputs. Each input is a bash command that can be used to run the program.

- Input: `bin/transplant -h`. Setting: 0x1 (`help` bit is set, other bits clear).
- Input: `bin/transplant -s`. Setting: 0x2 (`serialize` bit is set, other bits clear).
- Input: `bin/transplant -d -p /tmp -c`. Setting: 0xc (`deserialize` and `clobber` bits are set, other bits clear).
- Input: `bin/transplant -c -d`. Setting: 0x0. This is an error case because the specified argument ordering is invalid (`-c` is before `-d`). In this case `validargs` returns -1, leaving `global_options` unset.

Part 2: Serialized Data Format

`Transplant` uses a serialized data format that has been designed with some features to help detect whether or not the serialized data being read has been corrupted. The serialized data produced by "transplant" consists of a sequence of *records*. Each record starts with a fixed-format *header*, which specifies the *type* of the record and the total *length* of the record in bytes (including the header). Some types of records consist only of a header. Other types contain data that immediately follows the header.

Each record header consists of 16 bytes of data, having the following format:

> "Magic" (3 bytes): 0x0C, 0x0D, 0xED Type (1 byte) Depth (4 bytes) Size (8 bytes)

The first three bytes are the "magic sequence" of values 0x0C, 0x0D, 0xED. The presence of this sequence in each record helps a program reading the serialized data to detect if it has been corrupted: if a record header is expected, but the magic sequence is not seen, then the program can immediately terminate with an error report. Following the magic sequence is a single byte that specifies the type of the record. The possible record types are listed below. Following the type are four bytes that specify the depth in the tree for this record, as detailed below. The depth is specified as an unsigned 32-bit integer in *big-endian* format (most significant byte first). Following the type are 8 bytes that specify the size of the record as an unsigned 64-bit integer, also in big-endian format. The size of the record is the total number of bytes comprising the record, including the header bytes as well as any additional data after the header.

There are six different record types:

- `START_OF_TRANSMISSION` (type = 0)
- `END_OF_TRANSMISSION` (type = 1)
- `START_OF_DIRECTORY` (type = 2)
- `END_OF_DIRECTORY` (type = 3)
- `DIRECTORY_ENTRY` (type = 4)
- `FILE_DATA` (type = 5)

Serialized data produced by "transplant" always begins with a single `START_OF_TRANSMISSION` record and ends with a single `END_OF_TRANSMISSION` record. These records consist only of a header (and so their size is 16 bytes). The depth fields of these records contain the value 0.

A `START_OF_DIRECTORY` record indicates the beginning of data corresponding to a subdirectory and an `END_OF_DIRECTORY` entry indicates the end of data corresponding to a subdirectory. These records also consist only of a header. A `START_OF_DIRECTORY` record has a depth that is one greater than that of the `DIRECTORY_ENTRY` record that immediately precedes it. The corresponding subdirectory data comprises the consecutive sequence of records following the `START_OF_DIRECTORY` record which have a depth greater than or equal to that of the `START_OF_DIRECTORY` record. This sequence of records is required to be terminated by an `END_OF_DIRECTORY` record with the same depth, so that `START_OF_DIRECTORY` and `END_OF_DIRECTORY` records of each depth always occur in matched pairs, like parentheses, with the records giving the content of that subdirectory sandwiched in between.

A `DIRECTORY_ENTRY` record specifies the name of an entry within the current directory, along with metadata associated with that entry. There are `DIRECTORY_ENTRY` records both for regular files and for subdirectories. The metadata, which has a fixed length (see below), occurs immediately following the header. Following the metadata is a file name, whose length may vary. File names consist simply of a sequence of arbitrary bytes (with no null terminator). The length of the name can be determined by subtracting from the total size of the record the size of the header and the size of the metadata.

A `FILE_DATA` record specifies the content of the file whose name is given by the immediately preceding `DIRECTORY_ENTRY` record. The content consists of the sequence of bytes of data immediately following the `FILE_DATA` header. The length of this sequence of bytes can be determined by subtracting the size of the record header from the total size of the record.

The metadata contained in a `DIRECTORY_ENTRY` record consists of the following 12 bytes of data:

- 4 bytes of type/permission information (type `mode_t`, in big-endian format), as specifed for the `st_mode` field of the `struct stat` structure in the man page for the `stat()` system call.
- 8 bytes of size information (type `off_t`, in big-endian format), as specified for the `st_size` field of the `struct stat` structure.

Note that the header file `transplant.h` contains various definitions associated with the serialized data format described above. You should use the definitions from this header file where appropriate.

Part 3: **Required** Functions and Variables

The header file `const.h` contains specifications of functions that you **must** implement, as well as the declarations of several variables and arrays that you **must** use for their stated purposes. You must implement these functions because we will want to be able to test them

separately. You must use the variables declared here to store your data because you are **not** permitted to declare any arrays elsewhere, nor to use array subscripting `[ ]`, and you are **not** permitted to use any dynamic storage allocation, such as `malloc()`. The occurrence of array brackets `[ ]` in any of the code you write will result in your receiving a zero for the assignment.

The following summarizes the variables you must use and the functions you must implement.

```
/* Options info, set by validargs. */
int global_options;

/*
 * Buffer to hold a pathname component read from stdin.
 */
char name_buf[NAME_MAX];

/*
 * Buffer to hold the pathname of the current file or directory,
 * including the null byte terminator.  The path could be an absolute path
 * (starting with '/') or a relative path (not starting with '/').
 * relative path.
 */
char path_buf[PATH_MAX];

/*
 * Current length of the path in path_buf, not including the terminating
 * null byte.
 */
int path_length;
```

```
/**
 * @brief Validates command line arguments passed to the program.
 * @details This function will validate all the arguments passed to the
 * program, returning 0 if validation succeeds and -1 if validation fails.
 * Upon successful return, the selected program options will be set in the
 * global variable "global_options", where they will be accessible
 * elsewhere in the program.
 *
 * @param argc The number of arguments passed to the program from the CLI.
 * @param argv The argument strings passed to the program from the CLI.
 * @return 0 if validation succeeds and -1 if validation fails.
 * Refer to the homework document for the effects of this function on
 * global variables.
 * @modifies global variable "global_options" to contain a bitmap representing
 * the selected options.
 */
int validargs(int argc, char **argv);
```

```
/*
 * @brief  Initialize path_buf to a specified base path.
 * @details  This function copies its null-terminated argument string into
 * path_buf, including its terminating null byte.
 * The function fails if the argument string, including the terminating
 * null byte, is longer than the size of path_buf.  The path_length variable
 * is set to the length of the string in path_buf, not including the terminating
 * null byte.
 *
 * @param  Pathname to be copied into path_buf.
 * @return 0 on success, -1 in case of error
 */
int path_init(char *name);
```

```
/*
 * @brief  Append an additional component to the end of the pathname in path_buf.
 * @details  This function assumes that path_buf has been initialized to a valid
```

```
 * string.  It appends to the existing string the path separator character '/',
 * followed by the string given as argument, including its terminating null byte.
 * The length of the new string, including the terminating null byte, must be
 * no more than the size of path_buf.  The variable path_length is updated to
 * remain consistent with the length of the string in path_buf.
 *
 * @param  The string to be appended to the path in path_buf.  The string must
 * not contain any occurrences of the path separator character '/'.
 * @return 0 in case of success, -1 otherwise.
 */
int path_push(char *name);
```

```
/*
 * @brief  Remove the last component from the end of the pathname.
 * @details  This function assumes that path_buf contains a non-empty string.
 * It removes the suffix of this string that starts at the last occurrence
 * of the path separator character '/'.  If there is no such occurrence,
 * then the entire string is removed, leaving an empty string in path_buf.
 * The variable path_length is updated to remain consistent with the length
 * of the string in path_buf.  The function fails if path_buf is originally
 * empty, so that there is no path component to be removed.
 *
 * @return 0 in case of success, -1 otherwise.
 */
int path_pop();
```

```
/*
 * @brief Deserialize directory contents into an existing directory.
 * @details  This function assumes that path_buf contains the name of an existing
 * directory.  It reads (from the standard input) a sequence of DIRECTORY_ENTRY
 * records bracketed by a START_OF_DIRECTORY and END_OF_DIRECTORY record at the
 * same depth and it recreates the entries, leaving the deserialized files and
 * directories within the directory named by path_buf.
 *
 * @param depth  The value of the depth field that is expected to be found in
 * each of the records processed.
 * @return 0 in case of success, -1 in case of an error.  A variety of errors
 * can occur, including depth fields in the records read that do not match the
 * expected value, the records to be processed to not being with START_OF_DIRECTORY
 * or end with END_OF_DIRECTORY, or an I/O error occurs either while reading
 * the records from the standard input or in creating deserialized files and
 * directories.
 */
int deserialize_directory(int depth);
```

```
/*
 * @brief Deserialize the contents of a single file.
 * @details  This function assumes that path_buf contains the name of a file
 * to be deserialized.  The file must not already exist, unless the ``clobber''
 * bit is set in the global_options variable.  It reads (from the standard input)
 * a single FILE_DATA record containing the file content and it recreates the file
 * from the content.
 *
 * @param depth  The value of the depth field that is expected to be found in
 * the FILE_DATA record.
 * @return 0 in case of success, -1 in case of an error.  A variety of errors
 * can occur, including a depth field in the FILE_DATA record that does not match
 * the expected value, the record read is not a FILE_DATA record, the file to
 * be created already exists, or an I/O error occurs either while reading
 * the FILE_DATA record from the standard input or while re-creating the
 * deserialized file.
 */
int deserialize_file(int depth);
```

```
/*
 * @brief  Serialize the contents of a directory as a sequence of records written
 * to the standard output.
 * @details  This function assumes that path_buf contains the name of an existing
 * directory to be serialized.  It serializes the contents of that directory as a
 * sequence of records that begins with a START_OF_DIRECTORY record, ends with an
 * END_OF_DIRECTORY record, and with the intervening records all of type DIRECTORY_ENTRY.
 *
 * @param depth  The value of the depth field that is expected to occur in the
 * START_OF_DIRECTORY, DIRECTORY_ENTRY, and END_OF_DIRECTORY records processed.
 * Note that this depth pertains only to the "top-level" records in the sequence:
 * DIRECTORY_ENTRY records may be recursively followed by similar sequence of
 * records describing sub-directories at a greater depth.
 * @return 0 in case of success, -1 otherwise.  A variety of errors can occur,
 * including failure to open files, failure to traverse directories, and I/O errors
 * that occur while reading file content and writing to standard output.
 */
int serialize_directory(int depth);
```

```
/*
 * @brief  Serialize the contents of a file as a single record written to the
 * standard output.
 * @details  This function assumes that path_buf contains the name of an existing
 * file to be serialized.  It serializes the contents of that file as a single
 * FILE_DATA record emitted to the standard output.
 *
 * @param depth  The value to be used in the depth field of the FILE_DATA record.
 * @param size  The number of bytes of data in the file to be serialized.
 * @return 0 in case of success, -1 otherwise.  A variety of errors can occur,
 * including failure to open the file, too many or not enough data bytes read
 * from the file, and I/O errors reading the file data or writing to standard output.
 */
int serialize_file(int depth, off_t size);
```

```
/**
 * @brief Serializes a tree of files and directories, writes
 * serialized data to standard output.
 * @details This function assumes path_buf has been initialized with the pathname
 * of a directory whose contents are to be serialized.  It traverses the tree of
 * files and directories contained in this directory (not including the directory
 * itself) and it emits on the standard output a sequence of bytes from which the
 * tree can be reconstructed.  Options that modify the behavior are obtained from
 * the global_options variable.
 *
 * @return 0 if serialization completes without error, -1 if an error occurs.
 */
int serialize();
```

```
/**
 * @brief Reads serialized data from the standard input and reconstructs from it
 * a tree of files and directories.
 * @details  This function assumes path_buf has been initialized with the pathname
 * of a directory into which a tree of files and directories is to be placed.
 * If the directory does not already exist, it is created.  The function then reads
 * from from the standard input a sequence of bytes that represent a serialized tree
 * of files and directories in the format written by serialize() and it reconstructs
 * the tree within the specified directory.  Options that modify the behavior are
 * obtained from the global_options variable.
 *
 * @return 0 if deserialization completes without error, -1 if an error occurs.
 */
int deserialize();
```

Part 4: Implementation Notes

The required functions can be implemented without having to allocate any dynamic storage or any arrays other than the ones you have been given.

- For reading from `stdin`, you should use the C library function `getchar()`.
- For writing to `stdout`, you should use the C library function `putchar()`.
- To do file I/O, you should use the C standard I/O library. Specifically, you will need to use:
  - `FILE *f = fopen(path_buf, MODE)` - to open a file, either for reading (use "r" for `MODE`) or writing (use "w" for `MODE`).
  - `fclose(f)` - to close a file `f` that was previously opened with `fopen()`.
  - `fgetc(f)` - to read a single byte of data `c` from a file f.
  - `fputc(c, f)` - to write a single byte `c` of data to a file f.
  - `fflush(stdout)` - to cause data buffered for `stdout` to be "flushed" before your program terminates. These functions are document in Section 3 of the Linux manual pages. For example, `man 3 fopen` will show you the manual page for `fopen()`.
- To create a directory whose name is in `path_buf`, you should use `mkdir(path_buf, 0700)`. The `mode` argument of `0700` will create the directory with owner read, write, and execute permissions, and no others. Later, once the directory is populated, you will be resetting the mode according to the metadata in the `DIRECTORY_ENTRY` record. Use `man 2 mkdir` to read the manual page for the `mkdir()` system call.
- To traverse the entries of a directory, you should use `DIR *dir = opendir(path_buf);` to open the directory whose name is in `path_buf`, then `struct dirent *de = readdir(dir);` to iterate through the successive directory entries, then `closedir(dir)` once the end of the directory has been indicated by a `NULL` return from `readdir()`. From the pointer returned by `readdir()` you can access the name of the file (relative to the containing directory) using `de->d_name`. See `man 3 readdir` for the man page.
- To get metadata for a file or directory whose name is in `path_buf`, you should use `stat(path_buf, &stat_buf)`, where you have previously declared the `stat_buf` structure variable using `struct stat stat_buf;`. Use `man 2 stat` to read the manual page for the `stat()` system call. We are only interested in the `st_mode` and `st_size` fields of `stat_buf`, so you can ignore the others.
- To determine from the `mode` metadata field whether the metadata refers to a file or directory, use `S_ISREG(stat_buf.st_mode)` and `S_ISDIR(stat_buf.st_mode)` macros. The `S_ISREG()` and `S_ISDIR()` macros are documented in the man page concerning inodes, which you can read with `man 7 inode`.
- To set the access permissions on a file or directory, call `chmod(path_buf, mode & 0777)`, where `mode` is the value from the `mode` metadata field. We are only interested in the low-order nine bits of `mode` (i.e. the file permission bits), which is the reason for the `& 0777`. See `man 2 chmod` for the man page on `chmod()`.

Note that all of the above functions can return an error indication in various situations. Though it is tedious to do, a competent C programmer will check each call for an error return and take appropriate action if one occurs.

Part 5: Running the Completed Program

In serialization mode, the `transplant` program traverses a tree of files and directories and writes to `stdout`. In deserialization mode, the program reads from `stdin` and creates files and directories. As the data output to `stdout` or input from `stdin` is binary data, it will not be useful to enter input directly from the terminal or to display output directly to the terminal. Instead, the program can be run using *input and output redirection* as in the following example:

```
$ bin/transplant -s -p rsrc/testdir > outfile
$ echo $?
0
```

This will cause the serialized data output by the program (corresponding to the files and directories beneath directory `rsrc/testdir`) to be redirected to the file `outfile`.

 The `>` symbol tells the shell to perform "output redirection": the file `outfile` is created (or truncated if it already existed -- be careful!) and the output produced by the program on `stdout` is sent to that file instead of to the terminal.

 `$?` is an environment variable in bash which holds the return code of the previous program run. In the above, the `echo` command is used to display the value of this variable.

In the following example:

```
$ bin/transplant -d -p test_out < outfile
$ echo $?
0
```

serialized data is redirected *from* the file `outfile`, becoming the `stdin` seen by the program.

For debugging purposes, the contents of `outfile` can be viewed using the `od` ("octal dump") command:

```
$ od -t x1 outfile
0000000 0c 0d ed 00 00 00 00 00 00 00 00 00 00 00 00 10
0000020 0c 0d ed 02 00 00 00 01 00 00 00 00 00 00 00 10
0000040 0c 0d ed 04 00 00 00 01 00 00 00 00 00 00 00 1f
0000060 00 00 41 c0 00 00 00 00 00 00 10 00 64 69 72 0c
0000100 0d ed 02 00 00 00 02 00 00 00 00 00 00 00 10 0c
0000120 0d ed 04 00 00 00 02 00 00 00 00 00 00 00 23 00
0000140 00 81 a4 00 00 00 00 00 00 00 09 67 6f 6f 64 62
0000160 79 65 0c 0d ed 05 00 00 00 02 00 00 00 00 00 00
0000200 00 19 47 6f 6f 64 62 79 65 21 0a 0c 0d ed 04 00
0000220 00 00 02 00 00 00 00 00 00 00 22 00 00 81 a4 00
0000240 00 00 00 00 00 00 07 68 65 6c 6c 6f 31 0c 0d ed
0000260 05 00 00 00 02 00 00 00 00 00 00 00 17 48 65 6c
0000300 6c 6f 31 0a 0c 0d ed 03 00 00 00 02 00 00 00 00
0000320 00 00 00 10 0c 0d ed 04 00 00 00 01 00 00 00 00
0000340 00 00 00 21 00 00 81 a4 00 00 00 00 00 00 00 06
0000360 68 65 6c 6c 6f 0c 0d ed 05 00 00 00 01 00 00 00
0000400 00 00 00 00 16 48 65 6c 6c 6f 0a 0c 0d ed 03 00
0000420 00 00 01 00 00 00 00 00 00 00 10 0c 0d ed 01 00
0000440 00 00 00 00 00 00 00 00 00 00 10
0000453
```

This is the serialized output from the `rsrc/testdir` example discussed above (note that the `rsrc/testdir` test directory is included with the base code). The successive bytes at the beginning of the file have the hexadecimal values `0c 0d ed 00 00 00 ...`. You can readily identify this as the start of a record, due to the presence of the "magic sequence" `0c 0d ed`. The values in the first column indicate the offsets (in bytes) from the beginning of the file, specified as 7-digit octal (base 8) numbers.

 The `-t x1` flag instructs `od` to interpret the file as a sequence of individual bytes (that is the meaning of the "`1`" in "`x1`"), which are printed as hexadecimal values (that is the meaning of the "`x`" in "`x1`"). The `od` program has many options for setting the output format; another useful version is `od -bc`, which shows individual bytes of data as both ASCII characters and their octal codes. Refer to the "man" page for `od` for other possiblities.

If you use the above command with an `outfile` that is much longer, there would be so much output that the first few lines would be lost off of the top of the screen. To avoid this, you can *pipe* the output to a program called `less`:

```
$ od -t x1 outfile | less
```

This will display only the first screenful of the output and give you the ability to scan forward and backward to see different parts of the output. Type `h` at the `less` prompt to get help information on what you can do with it. Type `q` at the prompt to exit `less`.

Alternatively, the output of the program can be redirected via a pipe to the `od` command, without using any output file:

```
$ bin/transplant -s -p mydir | od -t x1 | less
```

Testing Your Program

Your program can be used with pipes to form a one-line command that will "transplant" a tree of files and directories from one place to another, by serializing files from a source directory and deserializing them into a different target directory:

```
$ bin/transplant -s -p mydir | bin/transplant -d -p otherdir
```

If the program is working properly, the files and directories under `mydir` will be copied to `otherdir`. To verify that the content of `mydir` and `otherdir` is now identical, you can use the `diff` command (use `man diff` to read the manual page):

```
diff -r mydir otherdir
```

For text files, `diff` will report the differences between the files in an understandable form, but for binary files, `diff` will just report that the files differ. To actually see the differences between binary files, you can use the `cmp` command to perform a byte-by-byte comparison of two files, regardless of their content:

```
$ cmp file1 file2
```

If the files have identical content, `cmp` exits silently. If one file is shorter than the other, but the content is otherwise identical, `cmp` will report that it has reached `EOF` on the shorter file. Finally, if the files disagree at some point, `cmp` will report the offset of the first byte at which the files disagree. If the `-l` flag is given, `cmp` will report all disagreements between the two files.

We can take this a step further and run an entire test without using any files:

```
$ cmp -l <(cat reffile) <(bin/transplant -s -p mydir)
$ echo $?
0
```

This compares the reference file `reffile` with the result of serializing the content of `mydir`.

 `<(...)` is known as process substitution. It is allows the output of the program(s) inside the parentheses to appear as a file for the outer program.

 `cat` is a command that outputs a file to `stdout`.

If the content of the reference file was identical to the output of `transplant`, `cmp` outputs nothing (as in this example).

Unit Testing

Unit testing is a part of the development process in which small testable sections of a program (units) are tested individually to ensure that they are all functioning properly. This is a very common practice in industry and is often a requested skill by companies hiring graduates.

 Some developers consider testing to be so important that they use a work flow called test driven development. In TDD, requirements are turned into failing unit tests. The goal is then to write code to make these tests pass.

This semester, we will be using a C unit testing framework called Criterion, which will give you some exposure to unit testing. We have provided a basic set of test cases for this assignment.

The provided tests are in the `tests/hw1_tests.c` file. These tests do the following:

- `validargs_help_test` ensures that `validargs` sets the help bit correctly when the `-h` flag is passed in.

- `validargs_serialize_test` ensures that `validargs` sets the serialize-mode bit correctly when the `-s` flag is passed in.
- `validargs_error_test` ensures that `validargs` returns an error when the `clobber` flag is specified together with the `-s` flag.
- `help_system_test` uses the `system` syscall to execute your program through Bash and checks to see that your program returns with `EXIT_SUCCESS`.

Compiling and Running Tests

When you compile your program with `make`, a `transplant_tests` executable will be created in your `bin` directory alongside the `transplant` executable. Running this executable from the `hw1` directory with the command `bin/transplant_tests` will run the unit tests described above and print the test outputs to `stdout`. To obtain more information about each test run, you can use the verbose print option: `bin/transplant_tests --verbose=0`.

The tests we have provided are very minimal and are meant as a starting point for you to learn about Criterion, not to fully test your homework. You may write your own additional tests in `tests/transplant_tests.c`. However, this is not required for this assignment. Criterion documentation for writing your own tests can be found here.

Hand-in instructions

TEST YOUR PROGRAM VIGOROUSLY!

Make sure your directory tree looks like this (possibly with additional files that you added) and that your homework compiles (you should be sure to try compiling with both `make clean all` and `make clean debug` because there are certain errors that can occur one way but not the other).

```
.
├── .gitlab-ci.yml
└── hw1
    ├── include
    │   ├── const.h
    │   ├── debug.h
    │   └── transplant.h
    ├── Makefile
    ├── rsrc
    │   ├── testdir
    │   │   ├── dir
    │   │   │   ├── goodbye
    │   │   │   └── hello1
    │   │   └── hello
    │   └── test_in.bin
    ├── src
    │   ├── main.c
    │   └── transplant.c
    └── tests
        └── hw1_tests.c
```

This homework's tag is: `hw1`

```
$ git submit hw1
```

When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly!